



**Computación en la Nube**

# **Informe Práctica 3**



**Universidad  
de La Laguna**

## Práctica 3

### 1. Elegir cualquier problema de tratamiento de imágenes e implementarlo en C/C++. Puede ser de tratamiento de vídeo.

Para este apartado he tratado de buscar un algoritmo secuencial que permita el tratamiento de una imagen. Tras buscar entre varios, me he decantado por el que podemos ver a continuación "<https://gist.github.com/OmarAflak/aca9d0dc8d583ff5a5dc16ca5cdda86a>".

El algoritmo comentado con anterioridad nos permite aplicar un filtro gaussiano que nos generará un desenfoque a una imagen. Esto lo podremos ver en la siguiente diapositiva:



### 2. Implementa una versión MPI para este algoritmo.

Tras entender el código secuencial me he dispuesto a modificarlo para crear una versión usando MPI. Los pasos para lograr esto han sido:

1. Inicializar la estructura de MPI.

```
// Inicializa la estructura de comunicación de MPI entre los procesos.
rc = MPI_Init(&argc, &argv);
// Determina el tamaño del grupo asociado con un comunicador
rc = MPI_Comm_size(MPI_COMM_WORLD, &size);
// Determina el rango (identificador) del proceso que lo llama dentro del comunicador seleccionado.
rc = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
tag = 100;
```

2. Dividir el programa en dos. Uno para si el proceso es el inicial y otro para el resto de los procesos.

3. Dividimos la imagen por el número de proceso. Esto nos ayudará a saber que sección deberá realizar cada proceso. Posteriormente, se enviará la imagen para que cada proceso pueda aplicar el filtrado.

```
// Calculamos los valores necesarios para poder aplicar el filtrado
int newImageHeightNode = userHeight/size;

// Enviamos la sección de la imagen a todos los procesos
int firstHeight = 0;
int finalHeight = 0;
for(int n = 1; n < size; n++){
    finalHeight = newImageHeightNode * n;
    for (int j = 0; j < 3; j++){
        for (int i = firstHeight; i < finalHeight; i++){
            rc = MPI_Send(&image[j][i][0], userWidth, MPI_DOUBLE, n, tag, MPI_COMM_WORLD);
        }
    }
    firstHeight += newImageHeightNode;
}
```

4. Cada proceso (incluido el principal) tendrá que realizar el filtrado de la sección que tiene asignada.

```
Image newImage = applyFilter(image, filter, 0, newImageHeightNode);
saveImage(newImage, "./src/0.png");
```

5. Por último, cada proceso enviará cada sección de imagen calculada y el principal las unificará para formar la Imagen final.

```
// Recogemos los valores y unificamos la Imagen
for(int n = 1; n < size; n++){
    Image newImageNode(3, Matrix(recvImageHeight, Array(newImageWidth)));
    for (int j = 0; j < 3; j++){
        for (int i = 0; i < recvImageHeight; i++){
            rc = MPI_Recv(&newImageNode[j][i][0], newImageWidth, MPI_DOUBLE, n, tag, MPI_COMM_WORLD, &status);
        }
    }
    if(n == 1)
    {
        finalImage = newImageNode;
    }
    else
    {
        finalImage = joinImage(finalImage, newImageNode);
    }
}

// Unimos la Imagen del nodo principal
finalImage = joinImage(finalImage, imageNodo0);

// Guardamos la Imagen
saveImage(finalImage, "./FinalImage.png");
```



Cabe resaltar que se ha modificado la función de aplicar el filtrado debido a que, era necesario que solo realizara una sección de la imagen y devuelva únicamente dicha sección. De igual forma, se ha intentado mejorar el código reduciendo un total de 10s.

Tras realizar todas las modificaciones, se ha compilado el código obteniendo los siguientes resultados que veremos a continuación. En caso de utilizar diferentes tamaños de kernel el programa tarda menos pero el filtro es menor.

### Programa 10x10 kernel

#### Ejecución

```


[ptondreau@ptondreau-1 MPI]$ make run
mpieexec -np 4 mpi_version sample3.png

--- Información de la Imagen---
height: 2112
width: 2816
filterHeight: 10
filterWidth: 10
newImageHeight: 2103
newImageWidth: 2807
newImageHeightNode: 525

Cargando...
Tiempo de ejecución 15 sec

```

#### Salida



### Programa kernel 5 x 5

#### Ejecución

```

[ptondreau@ptondreau-1 MPI]$ make run
mpieexec -np 4 mpi_version sample3.png

--- Información de la Imagen---
height: 2112
width: 2816
filterHeight: 5
filterWidth: 5
newImageHeight: 2108
newImageWidth: 2812
newImageHeightNode: 527

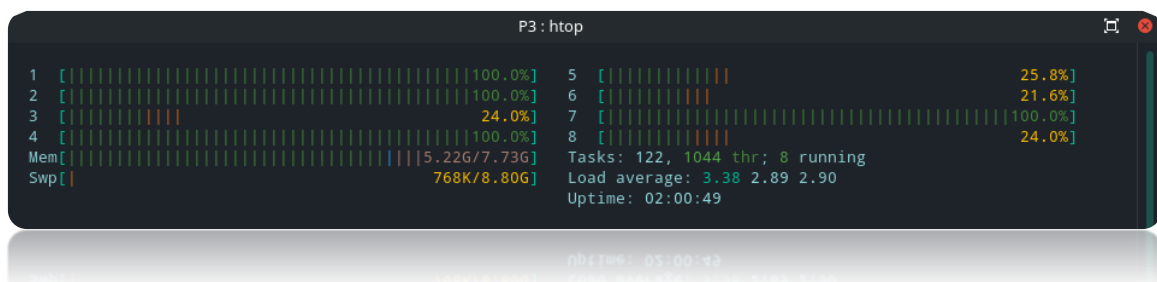
Cargando...
Tiempo de ejecución 8 sec

```

#### Salida



### HTOP



### 3. Desarrolla una versión OpenMP para este algoritmo.

Para poder crear una versión de OpenMP se ha cogido la versión secuencial y se ha modificado para que use todos los hilos posibles dentro de la máquina. Para lograr esto, se ha añadido lo siguiente a la función de aplicar el filtrado:

```
función que aplica un filtro dado un tamaño inicial y final. Además, solo devuelve ese trozo calculado
Image applyFilter(Image &image, Matrix &filter)
{
    assert(image.size() == 3 && filter.size() != 0);

    int height = image[0].size();
    int width = image[0][0].size();
    int filterHeight = filter.size();
    int filterWidth = filter[0].size();
    int newImageHeight = height - filterHeight + 1;
    int newImageWidth = width - filterWidth + 1;

    Image newImage(3, Matrix(newImageHeight, Array(newImageWidth)));

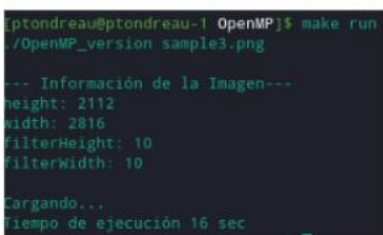

    int x = 0;

    for (int d = 0; d < 3; d++)
    {
        #pragma omp parallel for
        for (int i = 0; i < newImageHeight; i++)
        {
            for (int j = 0; j < newImageWidth; j++)
            {
                for (int h = 0; h < filterHeight; h++)
                {
                    for (int w = 0; w < filterWidth; w++)
                    {
                        newImage[d][i][j] += filter[h][w] * image[d][i+h][j+w];
                    }
                }
            }
        }
    }

    return newImage;
}
```

Tras añadir lo anterior, se ha probado el código obteniendo las diapositivas que veremos a continuación. En caso de utilizar diferentes tamaños de kernel, el programa tarda menos pero el filtro es menor

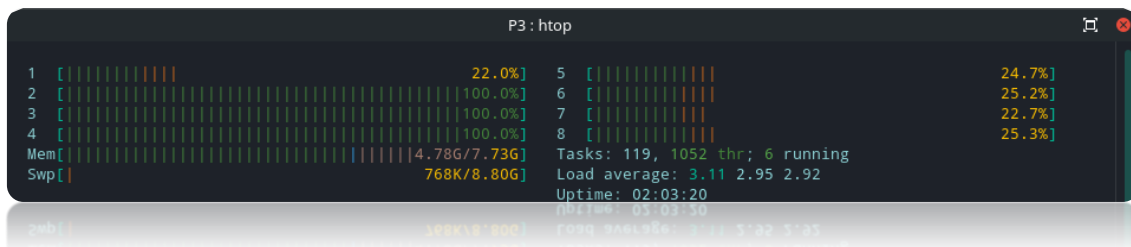
#### Programa 10x10 kernel

Ejecución	Salida
	
	

## Programa 5x5 kernel



## HTOP



## 4. Compara ambas versiones.

Entre ambas versiones existen diversas diferencias tanto en el tiempo de cómputo, como en la facilidad de la programación. Esto lo podremos ver a continuación:

### Tiempo de Computo

En cuanto al tiempo de cómputo, el código hecho con MPI es más eficiente que el realizado con OPENMP. Aunque en primera instancia, nos parezca lo contrario hay que tomar en cuenta que la opción de MPI debe descargar la imagen sobre cada proceso y, posteriormente, escribir y leer de un recurso compartido. Esta problemática se podría resolver intentando buscar un método más eficiente.

### Facilidad de programación

En cuanto a la facilidad de programación la opción OpenMP es mucho más sencillo y rápido de realizar. Esto es debido a que la gran parte de la complejidad de MPI ya se realiza de forma automática (envíos, dividir el problema,...).

## Conclusión

Por todo lo comentado con anterioridad, opino que ambas tecnologías tienen un uso para una circunstancia. Si se quiere primar la velocidad de programación se deberá de usar OpenMP. Sin embargo, si se quiere priorizar el tiempo de computo se deberá de usar MPI. Esto es debido principalmente a que MPI es más configurable. En cambio, OpenMp está más limitado.